

Práctica de Arquitectura y Organización de Computadores

Introducción a C - 2da parte

Segundo Cuatrimestre 2024

Arquitectura y Organización de Computadores
DC - UBA

Sobre la clase de hoy

- Alineación
- Estructuras
- Punteros
- Memoria dinámica
- Punteros a funciones
- Punteros a punteros
- Arrays multidimensionales
- Extras

Alineación

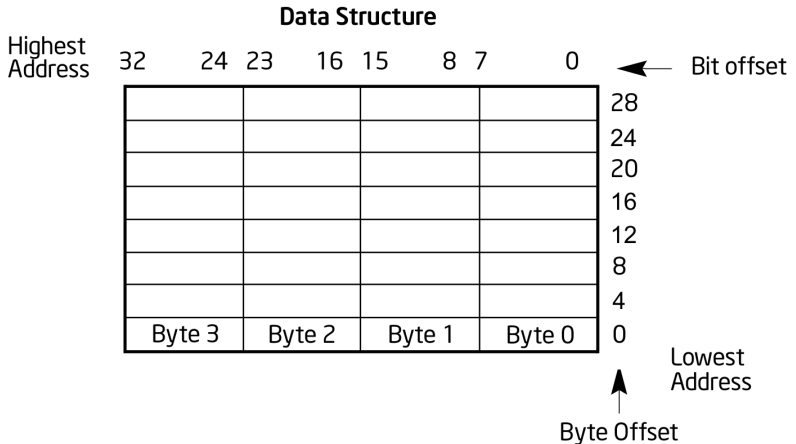
- Una palabra (*word*) es la cantidad de bits natural que el procesador puede manejar por operación.

- Una palabra (*word*) es la cantidad de bits natural que el procesador puede manejar por operación.
- La memoria es direccionable a **nivel de byte**, pero los accesos a memoria son más eficientes si se hacen en palabras.

- Una palabra (*word*) es la cantidad de bits natural que el procesador puede manejar por operación.
- La memoria es direccionable a **nivel de byte**, pero los accesos a memoria son más eficientes si se hacen en palabras.
- La alineación de los datos es importante para el rendimiento.


Alineación. Endianness


Little endian



¿Cómo cargamos i en AX (2 bytes)?

(Supongamos direcciones de 4 bytes)

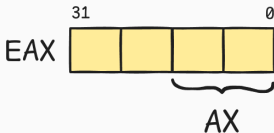
`int16_t i;` 

`char c;` 

`0x00000000` 

`0x00000004` 

`0x00000008` 



Alineación

Memoria

0x00000000



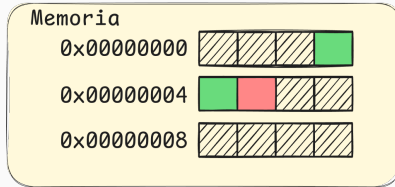
0x00000004



0x00000008



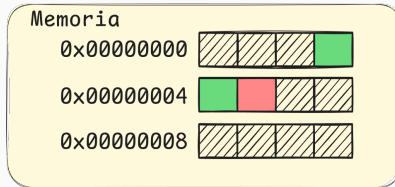
Alineación



```
mov EBX, 0x0  
mov ECX, [EBX]
```



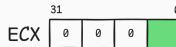
Alineación



```
mov EBX, 0x0  
mov ECX, [EBX]
```



```
shr ECX, 24
```



Alineación

Memoria

0x00000000



0x00000004



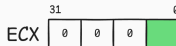
0x00000008



```
mov EBX, 0x0  
mov ECX, [EBX]
```



```
shr ECX, 24
```



```
mov EBX, 0x4  
mov EDX, [EBX]
```



Alineación

Memoria

0x00000000



0x00000004



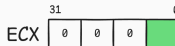
0x00000008



```
mov EBX, 0x0  
mov ECX, [EBX]
```



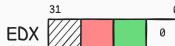
```
shr ECX, 24
```



```
mov EBX, 0x4  
mov EDX, [EBX]
```



```
shl EDX, 8
```



Memoria

0x00000000



0x00000004



0x00000008



```
mov EBX, 0x0  
mov ECX, [EBX]
```



```
shr ECX, 24
```



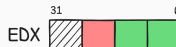
```
mov EBX, 0x4  
mov EDX, [EBX]
```



```
shl EDX, 8
```



```
or EDX, ECX
```



Alineación

Memoria

0x00000000



0x00000004



0x00000008



```
mov EBX, 0x0  
mov ECX, [EBX]
```



```
shr ECX, 24
```



```
mov EBX, 0x4  
mov EDX, [EBX]
```



```
shl EDX, 8
```



```
or EDX, ECX
```



```
mov AX, DX
```



- Vamos a decir que un dato **se encuentra alineado a N bytes** si su dirección de memoria es múltiplo de N.

- Vamos a decir que un dato **se encuentra alineado a N bytes** si su dirección de memoria es múltiplo de N.
- Cada tipo de dato tiene su propio *requerimiento de alineación*

- Vamos a decir que un dato **se encuentra alineado a N bytes** si su dirección de memoria es múltiplo de N.
- Cada tipo de dato tiene su propio *requerimiento de alineación*
- Por ejemplo, un entero de 32 bits (4 bytes) debe estar alineado a 4 bytes.

Alineación

Entero de 32 bits

`int32_t c;` 

`0x00000000`



`0x00000004`





`0x00000008`



Alineación

Variables desalineadas

`int8_t a;` 

`int16_t b;` 

`int32_t c;` 


`0x00000000` 


`0x00000004` 


`0x00000008` 

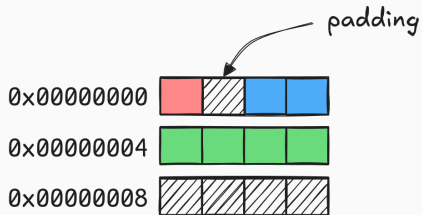
Alineación

Variables alineadas

`int8_t a;` 

`int16_t b;` 

`int32_t c;` 



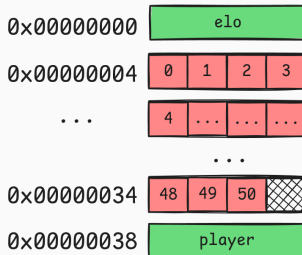
Estructuras

```
1  #define NAME_LEN 50
2
3  struct {
4      int elo;
5      char name[NAME_LEN + 1];
6      int ranking;
7  } player;
8
```

Layout en memoria

```
#define NAME_LEN 50

struct {
    int elo;
    char name[NAME_LEN + 1];
    int ranking;
} player;
```



Inicialización

```
1  #define NAME_LEN 50
2
3  struct player{
4      int elo;
5      char name[NAME_LEN + 1];
6      int ranking;
7  };
8
9  struct player player1 = { 2800, "Magnus Carlsen", 1 },
10 struct player player2 = { 2700, "Fabiano Caruana", 2 };
11
```

Definición de tipos

```
1 typedef float real_t;  
2 typedef int quantity_t;  
3 typedef unsigned long int size_t; // en <stdint.h>  
4 typedef uint32_t vaddr_t; // direccion virtual.  
5 typedef uint32_t paddr_t; // direccion fisica.
```

typedef

```
1  #define NAME_LEN 50
2
3  typedef struct {
4      int elo;
5      char name[NAME_LEN + 1];
6      int ranking;
7  } player_t;
8
9  player_t player1 = { 2800, "Magnus Carlsen", 1 },
10 player_t player2 = { 2700, "Fabiano Caruana", 2 };
11 player_t player3 = { .name = "Hikaru Nakamura",
12                     .ranking = 3,
13                     .elo = 2600 }; //forma alternativa
```

Operaciones

```
1  player_t magnus = { 2800, "Magnus Carlsen", 1 },
2  player_t faustino;
3
4  printf("Elo: %d\n", magnus.elo);
5  printf("Name: %s\n", magnus.name);
6  printf("Ranking: %d\n", magnus.ranking);
7
8  magnus.elo = 2700;
9  magnus.ranking--;
10
11 faustino = magnus; // copia de estructura
```

Como argumentos y valores de retorno

```
1  player_t get_player(void) {  
2      player_t player = { 3000, "Bobby Fischer", 1 };  
3      return player;  
4  }  
5  
6  void print_player(player_t player) {  
7      printf("Elo: %d\n", player.elo);  
8      printf("Name: %s\n", player.name);  
9      printf("Ranking: %d\n", player.ranking);  
10 }  
11
```

Array de estructuras

```
1  typedef struct {
2      char* pais;
3      int code;
4  } dials_code_t;
5
6  dials_code_t country_codes[] = {
7      {"Argentina", 54},
8      {"Brasil", 55},
9      {"Chile", 56},
10     {"Uruguay", 598}
11 };
12
13 printf("Código para Argentina: %d\n", country_codes[0].code);
```

```
1  dials_code_t country_codes[100] = {  
2      [0] = {"Argentina", 54},  
3      [1] = {"Brasil", 55},  
4      [2] = {"Chile", 56},  
5      [3].pais = "Uruguay", [3].code = 598,  
6      // ... el resto se inicializa en 0  
7  };  
8
```

Layout en memoria

- Cada campo está alineado con su tamaño. Se inserta padding para cumplir con los requerimientos de alineación.

Layout en memoria

- Cada campo está alineado con su tamaño. Se inserta padding para cumplir con los requerimientos de alineación.
- Una estructura siempre se alinea con el mayor de los requerimientos de alineación de sus campos.

Layout en memoria

- Cada campo está alineado con su tamaño. Se inserta padding para cumplir con los requerimientos de alineación.
- Una estructura siempre se alinea con el mayor de los requerimientos de alineación de sus campos.
- El tamaño de una estructura es tal que la dirección que sigue a la estructura tiene la alineación de la misma estructura. Se inserta padding si es necesario

Layout en memoria

- Cada campo está alineado con su tamaño. Se inserta padding para cumplir con los requerimientos de alineación.
- Una estructura siempre se alinea con el mayor de los requerimientos de alineación de sus campos.
- El tamaño de una estructura es tal que la dirección que sigue a la estructura tiene la alineación de la misma estructura. Se inserta padding si es necesario
- **Confusión típica:** no confundir el tamaño de una estructura (`sizeof`) con su requisito de alineación.

Ejemplos:

```
struct alumno {  
    char* nombre;  
    char comision;  
    int dni;  
};
```

```
struct alumno2 {  
    char comision;  
    char* nombre;  
    int dni;  
};
```

```
struct alumno3 {  
    char* nombre;  
    int dni;  
    char comision;  
} __attribute__((packed));
```

```
struct alumno {  
    char* nombre;    → 8  
    char comision;   → 1  
    int dni;         → 4  
};
```

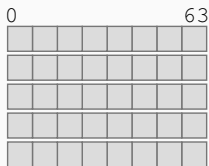
```
struct alumno2 {  
    char comision;   → 1  
    char* nombre;    → 8  
    int dni;         → 4  
};
```

```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```

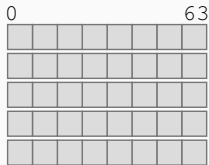
Ejemplos:

→ SIZE

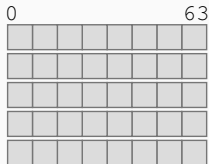
```
struct alumno {  
    char* nombre;    → 8  
    char comision;   → 1  
    int dni;         → 4  
};
```



```
struct alumno2 {  
    char comision;    → 1  
    char* nombre;     → 8  
    int dni;          → 4  
};
```



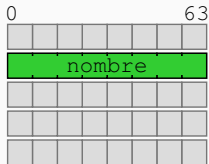
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



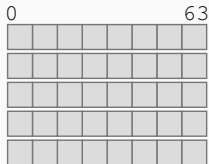
Ejemplos:

→ SIZE ⇒ OFFSET

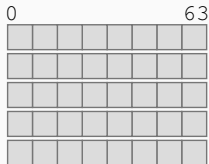
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1  
    int dni;         → 4  
};
```



```
struct alumno2 {  
    char comision;    → 1  
    char* nombre;     → 8  
    int dni;          → 4  
};
```



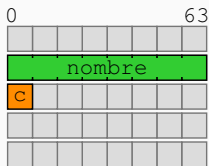
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



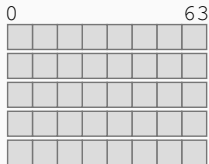
Ejemplos:

→ SIZE ⇒ OFFSET

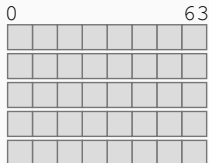
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4  
};
```



```
struct alumno2 {  
    char comision;    → 1  
    char* nombre;     → 8  
    int dni;          → 4  
};
```



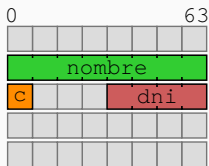
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



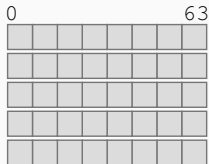
Ejemplos:

→ SIZE ⇒ OFFSET

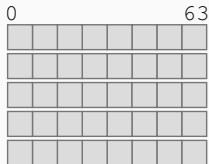
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};
```



```
struct alumno2 {  
    char comision;    → 1  
    char* nombre;     → 8  
    int dni;          → 4  
};
```



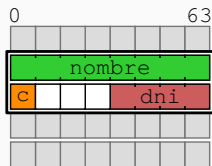
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



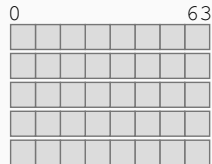
Ejemplos:

→ SIZE ⇒ OFFSET

```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;   → 1  
    char* nombre;    → 8  
    int dni;         → 4  
};
```



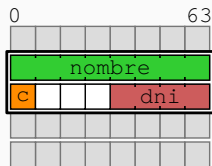
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



Ejemplos:

→ SIZE ⇒ OFFSET

```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;    → 1    ⇒ 0  
    char* nombre;     → 8  
    int dni;          → 4  
};
```



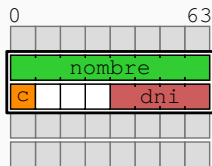
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



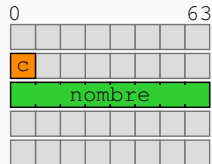
Ejemplos:

→ SIZE ⇒ OFFSET

```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;   → 1    ⇒ 0  
    char* nombre;    → 8    ⇒ 8  
    int dni;         → 4  
};
```



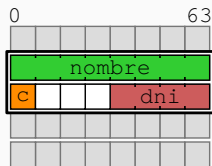
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



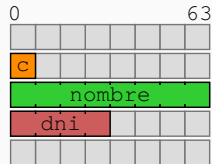
Ejemplos:

→ SIZE ⇒ OFFSET

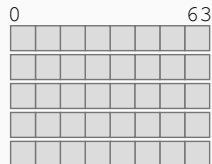
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;    → 1    ⇒ 0  
    char* nombre;     → 8    ⇒ 8  
    int dni;          → 4    ⇒ 16  
};
```



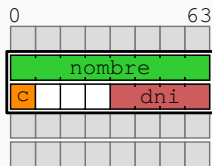
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



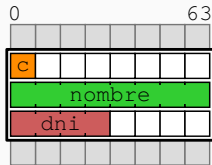
Ejemplos:

→ SIZE ⇒ OFFSET

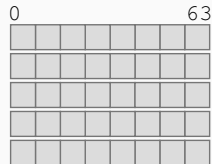
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;   → 1    ⇒ 0  
    char* nombre;    → 8    ⇒ 8  
    int dni;         → 4    ⇒ 16  
};                  ⇒ 24
```



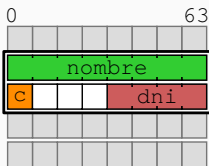
```
struct alumno3 {  
    char* nombre;    → 8  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



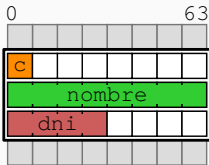
Ejemplos:

→ SIZE ⇒ OFFSET

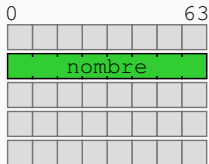
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;   → 1    ⇒ 0  
    char* nombre;    → 8    ⇒ 8  
    int dni;         → 4    ⇒ 16  
};                  ⇒ 24
```



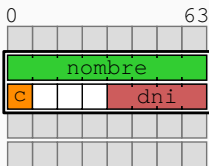
```
struct alumno3 {  
    char* nombre;    → 8    ⇒ 0  
    int dni;         → 4  
    char comision;   → 1  
} __attribute__((packed));
```



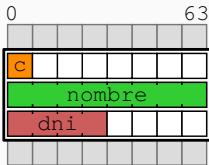
Ejemplos:

→ SIZE ⇒ OFFSET

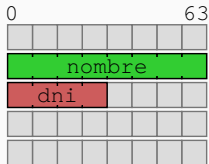
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;   → 1    ⇒ 0  
    char* nombre;    → 8    ⇒ 8  
    int dni;         → 4    ⇒ 16  
};                  ⇒ 24
```



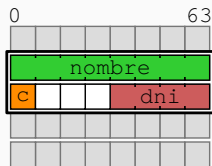
```
struct alumno3 {  
    char* nombre;    → 8    ⇒ 0  
    int dni;         → 4    ⇒ 8  
    char comision;   → 1  
} __attribute__((packed));
```



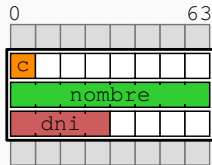
Ejemplos:

→ SIZE ⇒ OFFSET

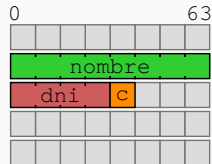
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;   → 1    ⇒ 0  
    char* nombre;    → 8    ⇒ 8  
    int dni;         → 4    ⇒ 16  
};                  ⇒ 24
```



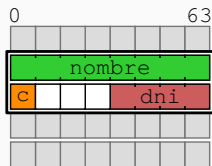
```
struct alumno3 {  
    char* nombre;    → 8    ⇒ 0  
    int dni;         → 4    ⇒ 8  
    char comision;   → 1    ⇒ 12  
} __attribute__((packed));
```



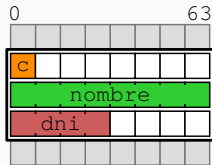
Ejemplos:

→ SIZE ⇒ OFFSET

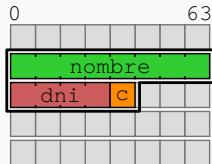
```
struct alumno {  
    char* nombre;    → 8    ⇒ 0  
    char comision;   → 1    ⇒ 8  
    int dni;         → 4    ⇒ 12  
};                  ⇒ 16
```



```
struct alumno2 {  
    char comision;   → 1    ⇒ 0  
    char* nombre;    → 8    ⇒ 8  
    int dni;         → 4    ⇒ 16  
};                  ⇒ 24
```



```
struct alumno3 {  
    char* nombre;    → 8    ⇒ 0  
    int dni;         → 4    ⇒ 8  
    char comision;   → 1    ⇒ 12  
} __attribute__((packed)); ⇒ 13
```



```
1 struct player{  
2     long elo;  
3     char name[21];  
4     int ranking;  
5 };
```

¿Cuál es el tamaño de la estructura?

```
1 struct player{  
2     long elo;  
3     char name[21];  
4     int ranking;  
5 };
```

¿Cuál es el tamaño de la estructura?

Rta: 40 bytes ($8 + 21 + 3(\text{Padding}) + 4 + 4(\text{Padding})$)

```
1 struct player{  
2     long elo;  
3     char name[21];  
4     int ranking;  
5 };
```

¿Cuál es el tamaño de la estructura?

Rta: 40 bytes ($8 + 21 + 3(\text{Padding}) + 4 + 4(\text{Padding})$)

¿Cuál es su requisito de alineación?

```
1 struct player{  
2     long elo;  
3     char name[21];  
4     int ranking;  
5 };
```

¿Cuál es el tamaño de la estructura?

Rta: 40 bytes ($8 + 21 + 3(\text{Padding}) + 4 + 4(\text{Padding})$)

¿Cuál es su requisito de alineación?

Rta: 8 bytes

```
1 struct padre {  
2     char c;  
3     struct {  
4         long l;  
5         short x;  
6     } hijo;  
7     int y;  
8 };
```

¿Cuál es el tamaño de la estructura padre?

```
1 struct padre {  
2     char c;  
3     struct {  
4         long l;  
5         short x;  
6     } hijo;  
7     int y;  
8 };
```

¿Cuál es el tamaño de la estructura padre?

Rta: 32 bytes $(1 + 7(\text{Padding}) + 16(\text{sizeof}(\text{hijo})) + 4 + 4(\text{Padding}))$


```
1 struct padre {  
2     char c;  
3     struct {  
4         long l;  
5         short x;  
6     } hijo;  
7     int y;  
8 };
```

¿Cuál es el tamaño de la estructura padre?

Rta: 32 bytes $(1 + 7(\text{Padding}) + 16(\text{sizeof}(\text{hijo})) + 4 + 4(\text{Padding}))$

¿Cuál es su requisito de alineación?

```
1  struct padre {  
2      char c;  
3      struct {  
4          long l;  
5          short x;  
6      } hijo;  
7      int y;  
8  };
```

¿Cuál es el tamaño de la estructura padre?

Rta: 32 bytes $(1 + 7(\text{Padding}) + 16(\text{sizeof}(\text{hijo})) + 4 + 4(\text{Padding}))$

¿Cuál es su requisito de alineación?

Rta: 8 bytes

Punteros

¿Qué es un puntero?

Es una variable que almacena una dirección de memoria.

¿Qué es un puntero?

Es una variable que almacena una dirección de memoria.



¿Qué es un puntero?

Es una variable que almacena una dirección de memoria.



La **clave** para comprender punteros es entender cómo se organiza la memoria en C.

Punteros. Tipos de memoria

Tipos de memoria	Scope	Lifetime
Global	El archivo completo	el tiempo de vida de la aplicación
Estática	La función (o bloque) donde está declarada	el tiempo de vida de la aplicación
Automática (local)	La función (o bloque) donde está declarada	Mientras la función (o bloque) esté en ejecución
Dinámica	Determinado por los punteros que referencian esta memoria	Hasta que la memoria sea liberada

Punteros

```
int x = 42;  
int *p = &x;
```

decimos que p "apunta" a x

x: 0x100



⋮

p: 0x130



Punteros. Rudimentos

```
1  #include <stdio.h>
2
3  int main(){
4      int x = 42;
5      int *p = &x;
6
7      printf("Dir de x: %p Valor: %d\n", (void*) &x, x);
8      printf("Dir de pi: %p Valor: %p\n", (void*) &p, (void*) p);
9      printf("Valor de lo que apunta p: %d\n", *p);
10 }
```

Dirección de x: 0x7ffc6cc507ac Valor: 42

Dirección de p: 0x7ffc6cc507b0 Valor: 0x7ffc6cc507ac

Valor de lo que apunta p: 42

Punteros. Rudimentos

```
1  #include <stdio.h>
2
3  int main(){
4      int x = 42;
5      int *p = &x;
6
7      printf("x: %d\n", x);
8      *p = 200;
9      printf("x: %d\n", x);
10 }
```

x: 42

x: 200

Punteros. NULL

```
1  int *pi = NULL; // equivalente a pi = 0;
2  if(pi) {
3      // pi no es NULL
4  }else {
5      // pi es NULL
6  }
```

- NULL es una macro que se define en `stddef.h` y es un puntero nulo.

Punteros. NULL

```
1  int *pi = NULL; // equivalente a pi = 0;
2  if(pi) {
3      // pi no es NULL
4  }else {
5      // pi es NULL
6  }
```

- NULL es una macro que se define en `stddef.h` y es un puntero nulo.
- Se usa para indicar que un puntero no apunta a ninguna dirección de memoria.

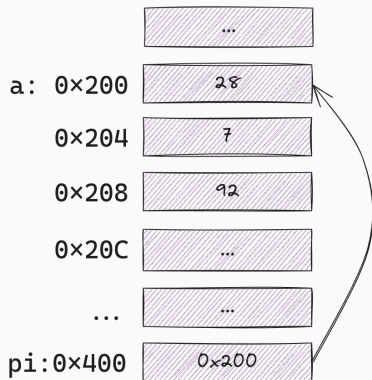
Punteros. NULL

```
1  int *pi = NULL; // equivalente a pi = 0;
2  if(pi) {
3      // pi no es NULL
4  }else {
5      // pi es NULL
6  }
```

- NULL es una macro que se define en `stddef.h` y es un puntero nulo.
- Se usa para indicar que un puntero no apunta a ninguna dirección de memoria.
- No se puede desreferenciar un puntero nulo.

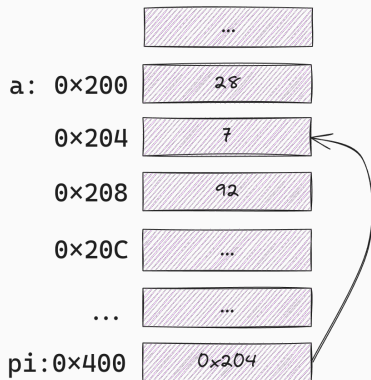
Aritmética de punteros

```
uint32_t a[] = {28,7,92};  
uint32_t *pi = a;  
  
printf("%d\n", *pi); //28
```



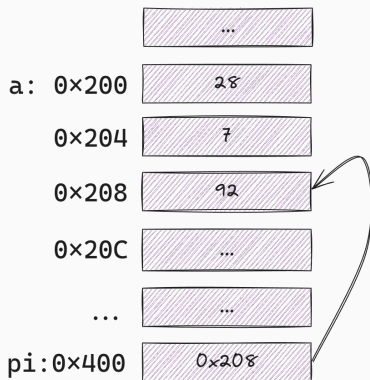
Aritmética de punteros

```
uint32_t a[] = {28,7,92};  
uint32_t *pi = a;  
  
printf("%d\n", *pi); //28  
pi += 1;  
printf("%d\n", *pi); //7
```



Aritmética de punteros

```
uint32_t a[] = {28,7,92};  
uint32_t *pi = a;  
  
printf("%d\n", *pi); //28  
pi += 1;  
printf("%d\n", *pi); //7  
pi += 1;  
printf("%d\n", *pi); //92
```



Punteros

```
1  #include <stdio.h>
2  void to_uppercase(char *str) {
3      while (*str){
4          if (*str >= 'a' && *str <= 'z') {
5              *str += ('A' - 'a');
6          }
7          str++;
8      }
9  }
10 int main() {
11     char text[] = "mayusculicenme!";
12     to_uppercase(text);
13     printf("Texto en mayúsculas: %s\n", text);
14 }
```

Texto en mayúsculas: MAYUSCULICENME!

Punteros y arrays

```
1  int arr[7] = {1,2,3,4,5,6,7};
2  int *p = arr;
3  // p apunta a la dirección del primer elemento del array
4
5  printf("%d\n", *p); // imprime 1
6  printf("%d\n", *(p+1)); // imprime 2
7  printf("%d\n", *(p+2)); // imprime 3
8  printf("%d\n", p[4]); // imprime 5
```

Punteros y arrays

- `p` es un puntero a `int`, por lo tanto, al hacer `p+1` estamos sumando 4 bytes a la dirección de memoria de `p`

Punteros y arrays

- `p` es un puntero a `int`, por lo tanto, al hacer `p+1` estamos sumando 4 bytes a la dirección de memoria de `p`
- al hacer `p+1` estamos apuntando a la dirección de memoria del segundo elemento del array

```
1  p[0] == *(p+0) == *p
2  p[1] == *(p+1)
3  ...
4  p[n] == *(p+n)
```

Punteros y arrays

- p es un puntero a `int`, por lo tanto, al hacer $p+1$ estamos sumando 4 bytes a la dirección de memoria de p
- al hacer $p+1$ estamos apuntando a la dirección de memoria del segundo elemento del array
- $p[4]$ es equivalente a $*(p+4)$, es decir, estamos accediendo al quinto elemento del array

```
1 p[0] == *(p+0) == *p
2 p[1] == *(p+1)
3 ...
4 p[n] == *(p+n)
```

Función swap

¿Esto funciona?

```
1 void swap(int a, int b) {  
2     int tmp = a;  
3     a = b;  
4     b = tmp;  
5 }  
6 int main() {  
7     int x = 10, y = 20;  
8     swap(x, y);  
9     printf("x: %d, y: %d\n", x, y);  
10 }
```

Punteros como argumentos

```
1 void swap(int *a, int *b) {  
2     int tmp = *a;  
3     *a = *b;  
4     *b = tmp;  
5 }  
6 int main() {  
7     int x = 10, y = 20;  
8     swap(&x, &y);  
9     printf("x: %d, y: %d\n", x, y);  
10 }
```

Memoria dinámica

¿Qué es la memoria dinámica?

Es un área de memoria que se asigna en tiempo de ejecución.

¿Qué es la memoria dinámica?

Es un área de memoria que se asigna en tiempo de ejecución.

Mucha de la potencia de los punteros radica en la capacidad de alocar memoria en tiempo de ejecución.

Los pasos básicos para trabajar con memoria dinámica son:

- Usar `malloc` (o `calloc`) para alocar memoria.

Los pasos básicos para trabajar con memoria dinámica son:

- Usar `malloc` (o `calloc`) para alocar memoria.
- Usar la memoria para lo que necesitamos.

Los pasos básicos para trabajar con memoria dinámica son:

- Usar `malloc` (o `calloc`) para alocar memoria.
- Usar la memoria para lo que necesitemos.
- Usar `free` para liberar la memoria.

Las declaraciones de las funciones `malloc` y `free` se encuentran en la librería `stdlib.h`

```
1 void *malloc(size_t size);  
2 void free(void *ptr);
```

Existen otras funciones como `calloc` y `realloc` que también se utilizan para alocar memoria.

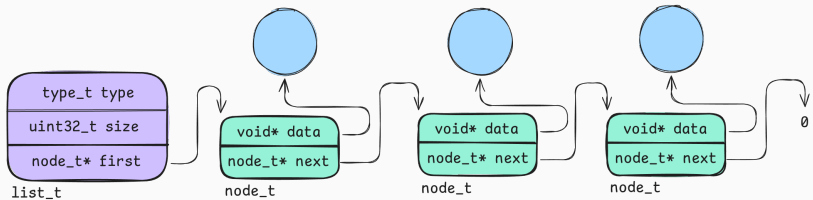
¿Esto funciona?

```
1  uint16_t *secuencia(uint16_t n){  
2      uint16_t arr[n];  
3      for(uint16_t i = 0; i < n; i++)  
4          arr[i] = i;  
5      return arr;  
6  }
```

Memoria dinámica

```
1  uint16_t *secuencia(uint16_t n){
2      uint16_t arr* = malloc(n * sizeof(uint16_t));
3      for(uint16_t i = 0; i < n; i++)
4          arr[i] = i;
5      return arr;
6  }
7  int main(){
8      uint16_t n = 10;
9      uint16_t *arr = secuencia(n);
10     for(uint16_t i = 0; i < n; i++)
11         printf("%d\n", arr[i]);
12     free(arr);
13 }
```


Lista simplemente enlazada



```
1 // type.h
2 typedef enum e_type {
3     TypeFAT32 = 0,
4     TypeEXT4 = 1,
5     TypeNTFS = 2
6 } type_t;
7
8 fat32_t* new_fat32();
9 ext4_t* new_ext4();
10 ntfs_t* new_ntfs();
11 fat32_t* copy_fat32(fat32_t* file);
12 ext4_t* copy_ext4(ext4_t* file);
13 ntfs_t* copy_ntfs(ntfs_t* file);
14 void rm_fat32(fat32_t* file);
15 void rm_ext4(ext4_t* file);
16 void rm_ntfs(ntfs_t* file);
```

Memoria dinámica

```
1 // list.h
2 #include "type.h"
3
4 typedef struct node {
5     void* data;
6     struct node* next;
7 } node_t;
8
9 typedef struct list {
10     type_t type;
11     uint8_t size;
12     node_t* first;
13 } list_t;
```

```
1 // list.h (cont)
2 list_t* listNew(type_t t);
3 void listAddFirst(list_t* l, void* data); //copia el dato
4 void* listGet(list_t* l, uint8_t i); //se asume: i < l->size
5 void* listRemove(list_t* l, uint8_t i); //se asume: i < l->size
6 void listDelete(list_t* l);
```

```
1 // list.c
2 list_t* listNew(type_t t) {
3     list_t* l = malloc(sizeof(list_t));
4     l->type = t; // l->type es equivalente a (*l).type
5     l->size = 0;
6     l->first = NULL;
7     return l;
8 }
```

`(*p_struct).field` es equivalente a `p_struct->field`

Memoria dinámica

```
1 void listAddFirst(list_t* l, void* data) {
2     node_t* n = malloc(sizeof(node_t));
3     switch(l->type) {
4         case TypeFAT32:
5             n->data = (void*) copy_fat32((fat32_t*) data);
6             break;
7         case TypeEXT4:
8             n->data = (void*) copy_ext4((ext4_t*) data);
9             break;
10        case TypeNTFS:
11            n->data = (void*) copy_ntfs((ntfs_t*) data);
12            break;
13    }
14    n->next = l->first;
15    l->first = n;
16    l->size++;
17 }
```

```
1 //se asume: i < l->size
2 void* listGet(list_t* l, uint8_t i){
3     node_t* n = l->first;
4     for(uint8_t j = 0; j < i; j++)
5         n = n->next;
6     return n->data;
7 }
```

Memoria dinámica

```
1 //se asume: i < l->size
2 void* listRemove(list_t* l, uint8_t i){
3     node_t* tmp = NULL;
4     void* data = NULL;
5     if(i == 0){
6         data = l->first->data;
7         tmp = l->first;
8         l->first = l->first->next;
9     }else{
10         node_t* n = l->first;
11         for(uint8_t j = 0; j < i - 1; j++){
12             n = n->next;
13             data = n->next->data;
14             tmp = n->next;
15             n->next = n->next->next;
16         }
17         free(tmp);
18         l->size--;
19         return data;
20     }
```

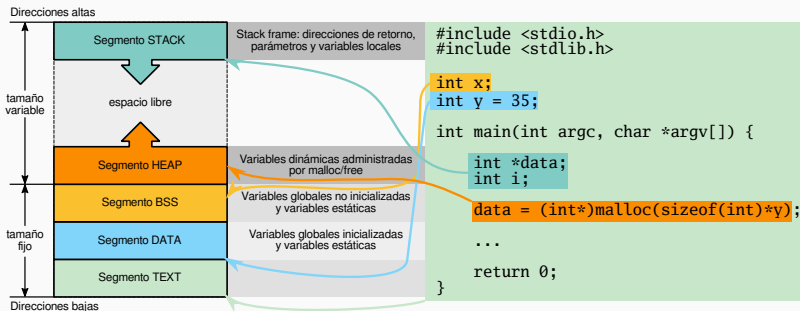

Memoria dinámica

```
1 void listDelete(list_t* l){
2     node_t* n = l->first;
3     while(n){
4         node_t* tmp = n;
5         n = n->next;
6         switch(l->type) {
7             case TypeFAT32:
8                 rm_fat32((fat32_t*) tmp->data);
9                 break;
10            case TypeEXT4:
11                rm_ext4((ext4_t*) tmp->data);
12                break;
13            case TypeNTFS:
14                rm_ntfs((ntfs_t*) tmp->data);
15                break;
16        }
17        free(tmp);
18    }
19    free(l);
20 }
```

```
1  int main(){
2      list_t* l = listNew(TypeFAT32);
3      fat32_t* f1 = new_fat32();
4      fat32_t* f2 = new_fat32();
5      listAddFirst(l, f1);
6      listAddFirst(l, f2);
7      listDelete(l);
8      rm_fat32(f1);
9      rm_fat32(f2);
10 }
```

Memoria dinámica

Imagen de memoria



Problemas con la memoria dinámica

- **Memory leak:** es la pérdida de memoria que no se libera nunca.

Problemas con la memoria dinámica

- **Memory leak:** es la pérdida de memoria que no se libera nunca.
- **Dangling pointer:** es un puntero que apunta a una dirección de memoria que ya no es válida.

Problemas con la memoria dinámica

- **Memory leak:** es la pérdida de memoria que no se libera nunca.
- **Dangling pointer:** es un puntero que apunta a una dirección de memoria que ya no es válida.
- **Double free:** es liberar la misma dirección de memoria dos veces.

Problemas con la memoria dinámica

- **Memory leak:** es la pérdida de memoria que no se libera nunca.
- **Dangling pointer:** es un puntero que apunta a una dirección de memoria que ya no es válida.
- **Double free:** es liberar la misma dirección de memoria dos veces.
- **Use after free:** acceder a memoria que ya fue liberada (caso particular de dangling pointer).

Problemas con la memoria dinámica

- **Memory leak:** es la pérdida de memoria que no se libera nunca.
- **Dangling pointer:** es un puntero que apunta a una dirección de memoria que ya no es válida.
- **Double free:** es liberar la misma dirección de memoria dos veces.
- **Use after free:** acceder a memoria que ya fue liberada (caso particular de dangling pointer).

La herramienta valgrind es muy útil para detectar estos problemas.

Punteros a función

Punteros a función

```
1 // Recibe un double y retorna un int
2 int (*f1)(double);
3
4 // Recibe un puntero a char y no retorna nada
5 void (*f2)(char*);
6
7 // Recibe dos enteros y retorna un puntero a double
8 double* (*f3)(int, int);
```

Punteros a función

```
1  #include <stdio.h>
2
3  void print_int(int x) {
4      printf("%d\n", x);
5  }
6
7  void pretty_print_int(int x) {
8      printf("Entero[%lu bits]: %d\n", sizeof(x)*8, x);
9  }
10
11 int main() {
12     void (*print)(int) = print_int;
13     print(42); // () desreferencia el puntero a función
14     print = pretty_print_int;
15     print(3);
16 }
```

Volviendo al ejemplo anterior...

```
1 // type.h (cont.)  
2 typedef void* (*funcCopy_t)(void*);  
3 typedef void (*funcRm_t)(void*);  
4 funcCopy_t getCopyFunction(type_t t);  
5 funcRm_t getRmFunction(type_t t);
```

Punteros a función

```
1 funcCopy_t getCpyFunction(type_t t) {
2     switch (t) {
3         case TypeFAT32: return (funcCopy_t) copy_fat32; break;
4         case TypeEXT4:  return (funcCopy_t) copy_ext4; break;
5         case TypeNTFS:  return (funcCopy_t) copy_ntfs; break;
6         default:        return NULL; break;
7     }
8 }
9
10 funcRm_t getRmFunction(type_t t) {
11     switch (t) {
12         case TypeFAT32: return (funcRm_t) rm_fat32; break;
13         case TypeEXT4:  return (funcRm_t) rm_ext4; break;
14         case TypeNTFS:  return (funcRm_t) rm_ntfs; break;
15         default:        return NULL; break;
16     }
17 }
```

Punteros a función

```
1 void listAddFirst(list_t* l, void* data) {
2     node_t* n = malloc(sizeof(node_t));
3     n->data = getCopyFunction(l->type)(data);
4     n->next = l->first;
5     l->first = n;
6     l->size++;
7 }
8 void listDelete(list_t* l){
9     node_t* n = l->first;
10    while(n){
11        node_t* tmp = n;
12        n = n->next;
13        getRmFunction(l->type)(tmp->data);
14        free(tmp);
15    }
16    free(l);
17 }
```

Punteros a Punteros

Punteros a punteros

¿Esto funciona?

```
1 void allocateArray(int *arr, int size, int value) {
2     arr = (int*)malloc(size * sizeof(int));
3     if(arr != NULL) {
4         for(int i=0; i<size; i++) {
5             arr[i] = value;
6         }
7     }
8 }
9 // Uso
10 int *vector = NULL;
11 allocateArray(vector,5,45);
12 for(int i = 0; i < 5; i++)
13     printf("%d\n", vector[i]);
14 free(vector);
```


Punteros a punteros

```
1 void allocateArray(int **arr, int size, int value) {
2     *arr = (int*)malloc(size * sizeof(int));
3     if(*arr != NULL) {
4         for(int i=0; i<size; i++) {
5             *(*arr +i) = value;
6         }
7     }
8 }
9 // Uso
10 int *vector = NULL;
11 allocateArray(&vector,5,45);
12 for(int i = 0; i < 5; i++)
13     printf("%d\n", vector[i]);
14 free(vector);
```

Arrays multidimensionales

Arrays multidimensionales

Consideraciones

- Un array multidimensional es un array de arrays.

Arrays multidimensionales

Consideraciones

- Un array multidimensional es un array de arrays.
- La memoria se almacena de forma contigua.

Consideraciones

- Un array multidimensional es un array de arrays.
- La memoria se almacena de forma contigua.
- La notación `a[i][j]` es equivalente a `*(a[i] + j)`.

Consideraciones

- Un array multidimensional es un array de arrays.
- La memoria se almacena de forma contigua.
- La notación `a[i][j]` es equivalente a `*(a[i] + j)`.
- La notación `a[i][j]` es equivalente a `*(a + i*COLS + j)`.

Arrays multidimensionales

Consideraciones

- Un array multidimensional es un array de arrays.
- La memoria se almacena de forma contigua.
- La notación `a[i][j]` es equivalente a `*(a[i] + j)`.
- La notación `a[i][j]` es equivalente a `*(a + i*COLS + j)`.
- El tipo de `a` es `int (*)[COLS]`.

Arrays multidimensionales

Consideraciones

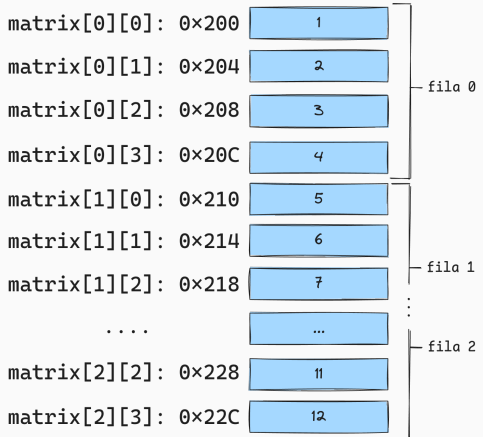
- Un array multidimensional es un array de arrays.
- La memoria se almacena de forma contigua.
- La notación $a[i][j]$ es equivalente a $*(a[i] + j)$.
- La notación $a[i][j]$ es equivalente a $*(a + i*COLS + j)$.
- El tipo de a es `int (*)[COLS]`.
- En memoria, los arrays multidimensionales se almacenan en orden de fila. (*Row-major order*)

Arrays multidimensionales

```
1  int matrix[3][4] = {
2      {1, 2, 3, 4},
3      {5, 6, 7, 8},
4      {9, 10, 11, 12}
5  };
6
7  // Todos estos prints imprimen 7
8  printf("matrix[1][2]: %d\n", matrix[1][2]);
9  printf("matrix[1][2]: %d\n", (*(matrix + 1) + 2));
10 printf("matrix[1][2]: %d\n", *((int*) matrix + 4*1 + 2));
11
12 m[0][3] = 100; // asigna 100 a la fila 0, columna 3
13
```

Arrays multidimensionales

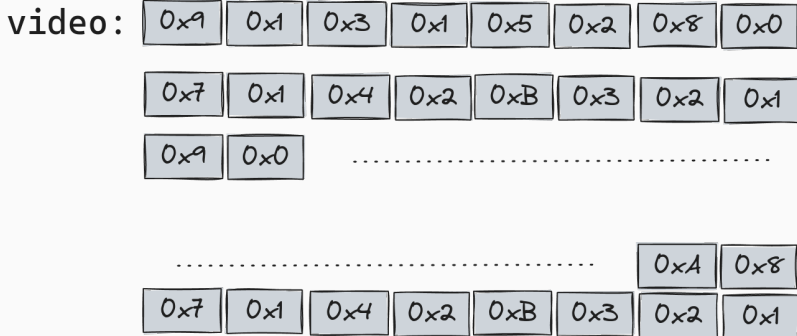
```
int matrix[3][4] = {  
    {1, 2, 3, 4},  
    {5, 6, 7, 8},  
    {9, 10, 11, 12}  
};
```



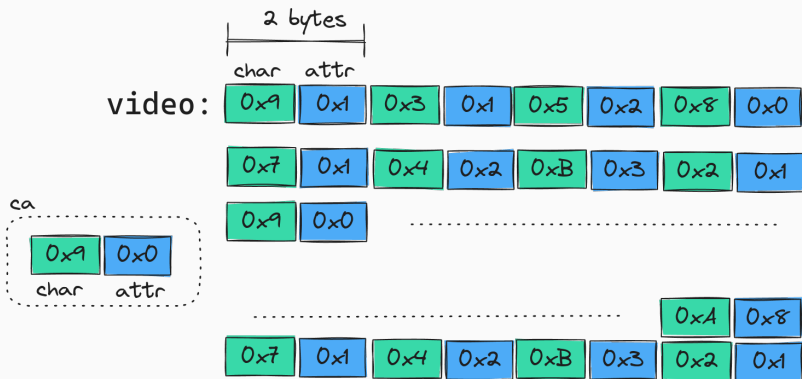
Arrays multidimensionales

```
1  #include <stdio.h>
2  int main() {
3      int matrix[3][4] = {
4          {1, 2, 3, 4},
5          {5, 6, 7, 8},
6          {9, 10, 11, 12}
7      };
8      // p apunta al int en la fila 0, columna 0
9      int *p = &matrix[0][0];
10
11     // reshape es un puntero a un array de 2 ints
12     int (*reshape)[2] = (int (*)[2]) p;
13
14     printf("%d\n", p[3]); // Qué imprime esta línea?
15     printf("%d\n", reshape[1][1]); // Qué imprime esta línea?
16 }
17
```

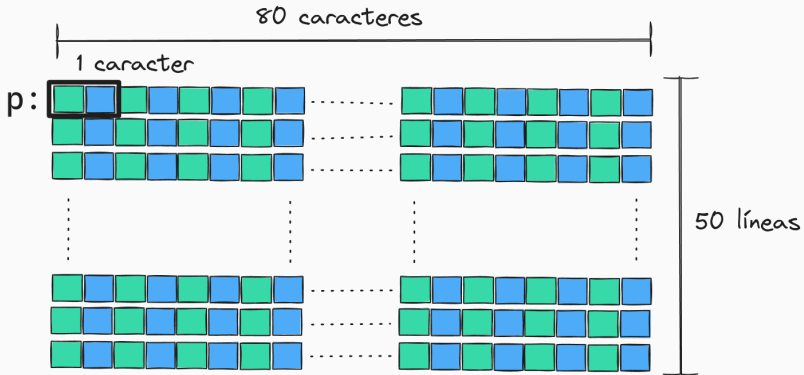
Array de array de structs



Array de arrays de structs



Array de array de structs



Array de arrays

```
1  #define VIDEO_COLS 80
2  #define VIDEO_FILS 50
3  // Cada posicion de memoria tiene 2 bytes
4  typedef struct ca_s {
5      uint8_t c; // caracter
6      uint8_t a; // atributos
7  } ca;
8  void screen_draw_layout(void) {
9      ca(*p)[VIDEO_COLS] = (ca(*)[VIDEO_COLS])VIDEO;
10     uint32_t f,c;
11     for (f = 0; f < VIDEO_FILS; f++) {
12         for (c = 0; c < VIDEO_COLS; c++) {
13             p[f][c].c = ' ';
14             p[f][c].a = 0x10;
15         }
16     }
17 }
```

Extras

tp0 magic

```
1 // ...
2 typedef struct{
3     uint64_t stock2: 11;
4     uint64_t stock1: 11;
5     uint64_t moneda2: 1;
6     uint64_t precio2: 12;
7     uint64_t moneda1: 1;
8     uint64_t precio1: 12;
9     uint64_t id2: 8;
10    uint64_t id1: 8;
11 }dupla_t;
12 reverse_ints_catalog();
13 dupla_t *dupla = (dupla_t *) catalogo;
14
15 for (int i = 0; i < 16; i++)
16 {
17     imprimir_producto(pfile, dupla[i].id1, dupla[i].precio1,
18                     dupla[i].moneda1, dupla[i].stock1);
19     imprimir_producto(pfile, dupla[i].id2, dupla[i].precio2,
20                     dupla[i].moneda2, dupla[i].stock2);
21 }
22 // ...
```

- *The C Programming Language*, Brian W. Kernighan y Dennis M. Ritchie.
- *C Programming: A Modern Approach*, K. N. King.
- *Understanding and Using C Pointers*, Richard Reese.
- *Effective C: An Introduction to Professional C Programming*, Robert C. Seacord.